New program constructs and semantics for reversible computation

Bill Stoddart and Campbell Ritchie
Teesside University UK

Abstract— We review our previously reported programming constructs for reversible computing which are based on the classical relational model. We note that, although mathematically elegant, the description of programming constructs is too abstract to capture the important concept of preference. To remedy this we formalise the temporal order of continuations, giving a richer semantics and yielding a new programming construct, whose implementation and application are discussed.

I. Introduction

Conceptualising computation as a reversible process allows us to formulate a simple mathematical semantics for sequential programming and define new programming constructs. In previous work [7], [9], [6] we have described these developments and the Reversible Virtual Machine (RVM) we use as an implementation platform for our ideas. Our approach exploits reversibility to handle backtracking and garbage collection. Backtracking is based on the idea of non-deterministic choice being interpreted as provisional choice, and the use of guarded commands which reverse execution when all available guards are false.

In this paper we look at the problem of expressing preference in choice within a formal semantics. In our previous work, and in the classical semantics of non-determinism, choice is described as a commutative connective, with an underlying model which views operations as relations between before and after states; choice is modelled as one before state being linked to two or more after states. This conceptual model is not rich enough to support a full description of preferential choice.

We have two motivations for wanting a richer model. Firstly, preference is important in programming, e.g. for expressing search heuristics. Secondly, the RVM has a choice construct implemented in such a way that the programmer can know which choice will be preferred, i.e. tried first. Thus we need a semantics of preference so as properly to describe our virtual machine.

We claim that the idea of preference is closely linked with that of continuations, where, by the continuation of a program S invoked within a larger program T, we mean the effect of the code from T which is executed following the termination of S. We shall sketch a semantics for the temporal order of continuations (TOC) which enables us to capture the idea of preference in choice. We shall also see how a new programming construct arises from our ideas, describe its implementation on the RVM and illustrate its possible application.

II. Preliminaries

We discuss programming constructs from both a theoretical high level guarded command language, to which we give the name RB0 [1] and from RVM-Forth the postfix language of the virtual machine to be used as a target for compilation of RB0 code. We begin with a review of some semantic features of RB0.

Within RB0, commands may be guarded; $g \rightarrow S$, pronounced "g guards S", will execute the program S if the condition g is true and will reverse execution otherwise.

The command $S \square T$, pronounced "S choice T" makes a provisional non-deterministic choice between the execution of S and T. If execution reverses back to this point, that choice will be revised and forward execution will start again with the other operation now selected for execution.

The construct $S \lozenge E$ represents the value expression E would take were it to be evaluated after the execution of S. It can be used as a term in the expression language of our executable language RB0. In operational terms it represents executing S, evaluating E, making a copy of E (which may be a data structure of arbitrary complexity) and then reversing the execution of S. The reverse execution uncomputes the original copy of E and reverses any side effects caused by the execution of S. $\lozenge$ is a low precedence connective, coming just above equivalence $\equiv$

We give two simple examples. $x:=2 \lozenge 10*x$ has the value 20. $x:=x+1 \lozenge 2*x$ has the value $2*(x+1)$.

Because of the presence of non-deterministic choice there may be more than one execution path through S, in which case $S \lozenge E$ can take multiple values. The set of such values is written $\{S \lozenge E\}$. This is also an allowable term in the expression language of RB0.

When $S \lozenge E$ yields multiple values we interpret the mathematical meaning of these using the Bunch Theory of E Hehner [3] according to which a bunch is the contents of a set, e.g. the contents of the set {1,2} is the bunch 1,2. The comma in the expression 1,2 is an operation, known as bunch union, rather than merely syntax. We define a guarded bunch expression $g \twoheadrightarrow E$ as equal to E when its guard g is true and equal to the empty bunch null otherwise.

The term $S \lozenge E$ plays two distinct roles. It can be a term in the extended expression language of RB0, and its

[1] B0 is the name of the run time language of the B formal development method, [1] which we adapt to reversible programs, naming the modified run time language RB0.

formal description is the basis of a "prospective value" semantics for the language, similar in expressive power to predicate transformer semantics.

We will now give rules for the values taken by terms of the form S ◊ E over the syntactic constructs of RB0. In the following, S and T are programs, and E and F are extended expressions, i.e expressions that may themselves include terms of the form S ◊ E. We have the following rules.

For skip. the program which does nothing, we have

skip ◊ E ≡ E

the value of E after executing skip is just its present value.

For assignment we have

x:=F ◊ E ≡ E[F/x].

Here, the term E[F/x] is the expression E rewritten with F replacing x. For example:

x:=2 ◊ x+1 ≡ (x+1)[2/x] ≡ 2+1

showing that the value of x+1 after executing x:=2 is 3.

For choice we have

S □ T ◊ E ≡ (S ◊ E) , (T ◊ E).

The prospective values after executing a choice are the bunch union of the prospective values associated with the individual choices. For example: x:=1 □ x:=2 ◊ x+10 is equivalent to 11,12.

For a guarded command we have

g → S ◊ E ≡ g ⇾ (S ◊ E) .

Here the symbol → is a program guard, being defined here, and the similar symbol ⇾ is the bunch guard, defined above. The prospective values associated with g → S are those associated with S if g is true, otherwise there are no such values.

For sequential composition of programs we have

S;T ◊ E ≡ S ◊ T ◊ E.

Note that to make sense of the term S ◊ T ◊ E the diamond operator has to be right associative, so that

S ◊ T ◊ E ≡ S ◊ (T ◊ E).

We can think of this as the value T ◊ E would take were S to be executed.

The interaction between choice, guards, and sequential composition which gives us the effect of backtracking is illustrated by the following simple example. We have a choice between assigning 1 or 2 to x, followed by a guarded command that required x to be 2. If x has been set to 1, this command will cause reverse execution, the other choice will be taken, and the final value of x will be 2. We do not explicitly describe this reversal of execution, but rather characterise the contribution of computation branches that are blocked as being null.

x:=1 □ x:=2 ; x=2 → skip ◊ x ≡ "by seq composition rule"
x:=1 □ x:=2 ◊ x=2 → skip ◊ x ≡ "by guard rule"
x:=1 □ x:=2 ◊ x=2 ⇾ (skip ◊ x) ≡ "by skip rule"
x:=1 □ x:=2 ◊ x=2 ⇾ x ≡ "by choice rule"
(x:=1 ◊ x=2 ⇾ x) , (x:=2 ◊ x=2 ⇾ x) ≡ "by assignment rule"
1=2 ⇾ 1 , 2=2 ⇾ 2 ≡ "by definition of bunch guard"
null , 2 ≡ "property of bunch union with the empty bunch"
2

A number of other constructs may be expressed within the prospective value scheme, including definition of local variables, choice from a set, and probabilistic choice. Search may be terminated on a given condition, and in particular S $◊_1$ E is the first result from a search. For a complete description see [6]. The implementation of S ◊ E and its related constructs is discussed in [8].

Conditionals and loop structures are not fundamental building blocks for our semantics, but are rather expressed through the use of choice, guard and sequencing commands. For example, for proof analysis we define:

IF g THEN S ELSE T END ≙ g → S □ ¬g → T

Indeed this is also an executable definition, though not an optimal one in our current implementation.

Reversibility of the RVM is implemented using a history stack. When the value of a variable is assigned in forward execution, the original value of the variable, the address of the variable, and the address of code that will restore the original value, are pushed onto the history stack. In this way forward execution compiles a simple program that will perform the corresponding reverse execution. Program execution is reversed by exchanging the history stack with the subroutine call stack (where subroutine return addresses are stored) and "returning" into the routine whose address was last deposited on the history stack. Each of the code fragments invoked in this way finds its arguments on the stack and, having consumed them, terminates with a return instruction that enters the next fragment of reverse execution code.

III. Choice, Preference, and Continuations

We are interested in two ways in which the characterisation of choice given above is to some extent inadequate. Firstly, although it allows us to express a provisional choice, it does not permit us to formally state which choice should be tried first. Secondly, the same semantics of choice is used within the B development method at a specification level to express "implementer's choice". That is, the choices an implementer is free to make whilst keeping within the brief of a specification. We can illustrate the difference between provisional and implementer's choice using the Knight's Tour chess problem, in which a program must find a sequence of moves which take a knight to each square of a chess board, visiting each only once. The specification of the program does not tell us which particular tour will be chosen, and we should be satisfied with any result that is a valid tour. That is implementer's choice. During execution of the program, provisional choices are made for moving the knight, and if a particular choice leads to an impasse, it will be revised. Covering the two choices by the same semantics is not completely unmanageable, but needs careful organisation

of the software development process, since discarding a possible implementer's choice (e.g. replacing S □ T by S during the program design process) still leaves us with a solution, but discarding a possible provisional choice may result in all solutions being lost. A full discussion is given in [11].

A. From recording sets of results to recording a temporally ordered sequence

With the aims of establishing a formal development method for reversible computation and exploiting reversibility to provide backtracking within the run time language of that method, we seek a semantics that can express preference and distinguish choices made during the design process from those made during program execution.

In effect, we are looking for a richer and more discriminating semantics than that provided by prospective values. In this regard, we recall that, thinking of S ◊ E as a term in an executable language, we are discarding information about the order in which results are recorded during its evaluation. This order could be retained if we recorded the results as a sequence. To see how this is achieved on the RVM and in RB0 we first look at the translation into RVM-Forth of the term {S ◊ E}.

For any RB0 program S or expression E let ⟦S⟧ or ⟦E⟧ respectively be its translation into RVM Forth. Let $⟦E⟧^T$ be the RVM Forth expression giving the type of E. Then we have the following translation schema for any term {S ◊ E}.

⟦{S ◊ E}⟧ = $⟦E⟧^T$ { <RUN ⟦S⟧ ⟦E⟧ RUN> }

For example { x:=1 □ x:=2 ◊ x+10 } would translate to:[2]

```
INT { <RUN
  <CHOICE 1 to x  []  2 to x  CHOICE>
  x 10 +
RUN> }
```

The result of evaluating this term will be the integer set {11,12}. Within the Forth code, each token in the above code is associated with an operation, as follows.

INT places a set on the stack which has the same type as the result. INT is the empty set of integers and acts as a template for the result set.

{ removes the set INT from the stack and opens a set construction for a set of the same type.

<RUN has no effect during forward execution, but converts reverse execution from within S to forward execution beyond the corresponding RUN>. Note that if reverse execution gets back to this point from within S all possible execution paths through S have been explored.

<CHOICE primes the history stack with the address of the second choice and an operation that will resume forward execution at this address.

[] separates choices in the choice construct. Code is compiled to skip over the rest of the choice construct. Were there to be more than one remaining choice, the history stack would also be primed at this point so that reverse execution caused forward execution to resume at the following choice.

CHOICE> marks the end of the choice structure. Code for each choice, other than the last, is terminated by a branch to this point, and the Forth compiler, which operates in a single pass, resolves these branches at this point.

RUN> adds a copy of the value of E to the set under construction (the original value found for E will be uncomputed by reverse execution) and reverses execution so that any further results can be found. Control will be passed beyond this point when reverse execution eventually succeeds in getting back to <RUN, indicating there are no more provisional choices to be explored.

} terminates the set construction, leaving (a pointer to) the final value of the set on the stack.

To consider how this schema might be amended so that the results generated by S are recorded as a sequence, we look first at the RVM-Forth expression of set extension and sequence extension. In the postfix notation of RVM-Forth the set extension {10,20} is written as:

```
INT { 10 , 20 , }
```

The comma in this code is an operation rather than syntax. It consumes the top stack value and adds it to the set under construction. This is why we have a comma after the 20.

A sequence with these same elements and with 10 occurring before 20 is written as:

```
INT [ 10 , 20 , ]
```

The comma in this sequence extension expression is an operation which consumes the top stack item and appends it to the sequence under construction. The commas for set and sequence construction are different operations, recognised as such by the Forth compiler setting a different "context" at the start of set or sequence extension expressions, resulting in the search for Forth "words" (tokens) being made on a different sequence of word-lists.

In this spirit we define, within the context of sequence extension, the operations <RUN and RUN>, similar to those used for set extensions but operating to construct a sequence of all values produced by an exhaustive exploration of S.

For any postfix program $\mathscr{F}$, leaving a single value of postfix type T,

T [<RUN $\mathscr{F}$ RUN>]

will leave the sequence of possible values generated by $\mathscr{F}$ by following different choices. The sequence will contain repeated values and the values it records will be held in the order in which they are generated. The range of the recorded sequence will be precisely the set of values generated by

T { <RUN $\mathscr{F}$ RUN> }

[2]We have omitted translation schemas for choice, assignment and arithmetic.

B. Guarded command language semantics for a language with preferential and implementer's choice

We now introduce a guarded command language structure to exploit the RVM-Forth sequence descriptions described in the previous section. We then provide a translation schema from guarded command language to RVM-Forth to show how the two descriptions are related.

The choice connective $\square$ which we have used in our guarded command language so far will no longer be a primitive of the language. It will be replaced, for the moment, by two different symbols. $S \rhd T$ will express preferential choice, representing a command that will first try S and later try T if the choice of S does not lead to a solution. $S \sqcap T$ will represent implementer's choice, used in specifications to express abstraction and used in the semantics of code where some details of implementation have been left to the compiler writer. We also give an unbounded version of implementer's choice. $\mathrm{var}\ v \bullet S$ introduces a local variable v which can non-deterministically take any value from its type and may be used in S, though not in E. Rather surprisingly, this construct has a use in describing implementations; if v is assigned a new value within S before being used in any expression, the effect of the non-determinism is removed, and the resulting description serves us as a semantics for local variables.

For any sequence expression E and program S we define the "nabla term" $S \nabla E$ over the syntactic structure of the language as follows:

The rule for skip is:

$\mathrm{skip} \nabla E \equiv E.$

For example $\mathrm{skip} \nabla \langle x \rangle \equiv \langle x \rangle.$

For assignment, recalling that E[F/x] represents a rewrite of expression E with expression F replacing x

$x := F \nabla E \equiv E[F/x]$

For example $x:=x+1 \nabla \langle x \rangle \equiv \langle x+1 \rangle$

For a guarded command, $g \rightarrow S$, the result of $g \rightarrow S \nabla E$ will be $S \nabla E$ if g is true, and otherwise will be an empty sequence, indicating that there are no possible continuations.

$g \rightarrow S \nabla E \equiv g \rightarrow (S \nabla E), \neg g \rightarrow \langle\rangle$

For example $x=1 \rightarrow \mathrm{skip} \nabla \langle x \rangle \equiv x=1 \rightarrow \langle x \rangle, x \neq 1 \rightarrow \langle\rangle$. Note that a key difference between the PV semantics of $S \Diamond E$ and the TOC semantics of $S \nabla E$ is that in PV semantics infeasibility leads to an empty bunch of after states, and in TOC semantics infeasibility leads to an empty sequence of continuations.

For preferential choice we capture the idea of preference through the order given to the possible continuations.

$S \rhd T \nabla E \equiv (S \nabla E)^\frown(T \nabla E)$

For example $x:=1 \rhd x:=2 \nabla \langle x \rangle \equiv \langle 1 \rangle^\frown\langle 2 \rangle \equiv \langle 1,2 \rangle$

Implementer's choice is represented as a bunch of possible sequences.

$S \sqcap T \nabla E \equiv (S \nabla E),(T \nabla E)$

For example $x:=1 \sqcap x:=2 \nabla \langle x \rangle \equiv \langle 1 \rangle,\langle 2 \rangle$ Note, however, that implementer's choice is a specification construct, supposed to be removed from a program during the development process. Nabla terms with non-deterministic choice are not executable.

For sequential composition, and noting that ∇ associates to the right:

$S ; T \nabla E \equiv S \nabla T \nabla E$

For example $x:=1;x:=x+1 \nabla \langle x \rangle \equiv x:=1 \nabla \langle x+1 \rangle \equiv \langle 2 \rangle$

Unbounded implementer's choice yields a (possibly infinite) bunch of continuations.

$\mathrm{var}\ v \bullet S \nabla E \equiv \oint v \bullet (S \nabla E)$

Here we have the restriction that v must not occur free in E. This restriction is not imposed by the form of the definition, since as far as the bunch comprehension expression is concerned, E is within the scope of v. However, it becomes clear when we consider that the aim of these rules is to describe all programs in our language by giving their TOC effect over the syntactic constructs of the elemental semantic components of the language (where we note that conditional statements and loops are composite constructs). Thus in this rule we are describing the TOC effect of the language construct $\mathrm{var}\ v \bullet S$.

We have omitted from these rules the purely specification construct of precondition, giving the rule for when a program S can be used, since its inclusion would require additional bunch theory concepts.

With these rules we can show that:

$x:=10 \rhd x:=20 \nabla \langle x \rangle \equiv \langle 10,20 \rangle$

and

$x:=10 \rhd x:=20 ; x=20 \rightarrow \mathrm{skip} \nabla \langle x \rangle \equiv \langle 20 \rangle$

In the second example, the preferred choice is x:=10. This subsequently leads to an impassable guard, causing the choice to be revised. We can think of $S \nabla \langle x \rangle$ as representing the sequence of values of x which program S will offers to following code when forced to backtrack from succeeding code. We can also think of it as a term in an extended expression language which will yield that sequence of values.

A perceptive reviewer asked why preferential choice sometimes only activates its less preferred option when forced to backtrack from the first, and sometimes executed both options. The answer lies in the previous paragraph - all choices within S are activated where it occurs in the term $S \nabla E$.

IV. Example programs and their RVM translations

We give three examples programs which exploit preferential choice and nabla terms, along with their translations into Forth.

We first give a schema for translating the RB0 term $S \nabla \langle E \rangle$.

We again use the brackets $[\![..]\!]$ to represent the translation of the program or expression within the brackets

into RVM-Forth, and we represent the postfix expression giving the type of E as $[\![E]\!]^T$.The translation is given by:

$$[\![S \nabla \langle E \rangle]\!] \equiv [\![E]\!]^T [\text{ <RUN } [\![S]\!] [\![E]\!] \text{ RUN> }]$$

Note that although nabla terms are formally defined for any sequence expression, non-unit sequences occur only within semantic analysis, so are not candidates for translation to RVM code.

A. A simple backtracking parser

Our first example program is a backtracking parser for a simple language of expressions containing only identifiers and addition and multiply symbols. We use preference to express precedence. The text to be parsed is taken from the input stream. Id, Otimes and Oplus attempt to parse an identifier, a multiply symbol and an addition symbol from the input stream. If this is possible they update the input stream pointer, otherwise they enter reverse execution.

We assume an input stream variable instream accessed as an array, and an input stream pointer i Also a set id_set containing the set of characters used as the valid identifiers for our language.

Here are the RB0 definitions

Id ≙ instream(i) ∈ id_set → i:=i+1
Oplus ≙ instream(i) = '+' → i:=i+1
Otimes ≙ instraim(i) = '∗' → i:=i+1

These are translated to RVM-Forth as

```
: Id instream i + C@ id-set IN --> i 1+ to i ;
: Oplus instream i + C@ [CHAR] + =  --> i 1+ to i ;
: Otimes instream i + C@ [CHAR] * =  --> i 1+ to i ;
```

We now consider the parsing operations T and E T parses expressions that contain no addition symbols, and E is the complete expression parser. Their RB0 definitions are.

T ≙ (Id ; Otimes ; T) ⊳ Id
E ≙ (T ; Oplus ; E) ⊳ T

These translate to the RVM-Forth as

```
: T <CHOICE Id Otimes RECURSE [] Id CHOICE> ;
: E <CHOICE T Oplus RECURSE [] T CHOICE> ;
```

B. Calculation of a probability distribution

As an example of a program that uses a nabla term, consider the calculation of frequencies of possible scores obtained by summing the values of three dice. We have an initialisation to record the scores associated with each of the 6^3 possible outcomes as a sequence, and an operation to interrogate the sequence and tell us the number of entries in the sequence that correspond to a given score. We assume a constant die equal to the set 1..6 and a variable scores which will record the sequence of possible scores resulting from the 6^3 possible outcomes that can arise from throwing three dice. The initialisation code is:

var x_1, x_2, x_3 • scores := (x_1 :∈ die; x_2 :∈ die; x_3 :∈ die ∇ $\langle x_1+x_2+x_3 \rangle$)

This translates to the following Forth operation, which it is convenient to name Init (the initialisation of an RB0 "abstract machine" being anonymous.)

```
: Init ( -- ) (: :)
  NULL VALUE x1  NULL VALUE x2  NULL VALUE x3
  INT [ <RUN
   DIE CHOICE to x1 DIE CHOICE to x2  DIE CHOICE to x3
    x1 x2 + x3 +
  RUN> ] to scores ;
```

Or, with optimisation, to

```
: Init ( -- )
  INT [ <RUN
    DIE CHOICE DIE CHOICE + DIE CHOICE +
  RUN> ] to scores ;
```

The scores sequence is held as a function which maps each position in the sequence to its value. The frequency of a particular score is found by range restricting this function to the required score, and taking the cardinality of the resulting set. ⊳ represents range restriction.

f ← freq(n) ≙ f := card(scores ⊳ n)

This translates to the following definition in the RVM

```
: freq ( n -- f ) (: n :) scores INT { n , } ⊳ CARD ;
```

C. Checking for redundant paths in backtracking search

A more general application of nabla terms is to check backtracking searches for redundant paths to solutions, that is to ensure they have only one way of finding each result. That will indeed be the case if

$$\text{card}(\{S \lozenge E\}) = \text{card}(S \nabla \langle E \rangle)$$

V. Related Work

A 1973 paper of M Zelkowitz [10] is the first communication to propose reversible execution as a means of controlling backtracking. Zelkowitz added a RETRACE command to PL1 allowing execution to reverse a given number of steps or until a certain condition was met. The feature is illustrated with a tree walking algorithm.

In "A Generalization of Dijkstra's Calculus" [5] Nelson discusses backtracking and clairvoyant choice, equivalent to the provisional choice of our PV semantics but analysed using weakest pre-conditions. He also introduces a form of biased choice S ⊞ T which will choose S unless there is no feasible path through S (no path not blocked by a false guard) in which case it chooses T. This construct differs from our preferential choice in that it is not concerned with what happens after S has terminated - it cannot revise its preference after backtracking from the continuation of S. Nelson uses his biased choice in the specification of a simple parsing algorithm, similar to our example but not executable. The first researcher to suggest that guards and choice rather than conditionals and loop statements were the fundamental building blocks of a sequential language

was R Hehner, in an unpublished but well known technical report, the history of which is given in [4]. Henry Baker [2] mentions a number of ways of exploiting reversibility that are related to this work, including speculative and subjunctive computation. Note that we can consider the term S ◊ E in a subjunctive sense as the values E might take were the computation S to be performed.

VI. Conclusions and Future Work

We propose a reversible virtual machine, the RVM, as a target platform for compiling reversible guarded command languages. For discussion purposes we adapt B0, the implementation level language of the B Method, to give RB0, a reversible guarded command language used as a vehicle for our discussions.

In this paper we add a preferential choice operator and a new programming construct that records the sequence of values an expression can take following a computation, recorded in the order they will be offered to the computation's continuation. This new programming structure, a nabla Term, is described and illustrated by a program that calculates a probability distribution. The semantic rules for nabla Terms give us the required formal description for preferential choice, whose use is illustrated by a simple backtracking parser.

Future work includes both theoretical and practical investigations. Since we now have two semantics, based on prospective values and the temporal order of continuations, there is an obvious obligation to prove these are consistent. For use of TOC semantics in program refinement, we need to develop a theory of proof obligations in the style of the B-Method, and which tell us what needs to be proved to show an implementation is consistent with its specification, which may be described at a far more abstract (and non-executable) level.

Ongoing practical work includes the development of a compiler for a variant of the theoretical high level guarded command language RB0. This language will include terms of the form S◊E and S▽E within its syntax of extended expressions.

Acknowledgements. We thank the referees for their helpful remarks and careful reading of the paper. We thank Steve Dunne, Frank Zeyda and Robert Lynas for many interesting conversations.

References

[1] J-R Abrial. The B Book. Cambridge University Press, 1996.
[2] H G Baker. The Thermodynamics of Garbage Collection. In Y Bekkers and Cohen J, editors, Memory Management: Proc IWMM'92, number 637 in Lecture Notes in Computer Science, 1992.
[3] E C R Hehner. A Practical Theory of Programming. Springer Verlag, 1993. Latest version available on-line.
[4] E. R. Hehner. Retrospective and Prospective for Unifying Theories of Programming. In S. E. Dunne and W Stoddart, editors, UTP2006 The First International Symposium on Unifying Theories of Programming, number 4010 in Lecture Notes in Computer Science, 2006.
[5] Greg Nelson. A Generalization of Dijkstra's Calculus. ACM Transactions on Programming Languages and Systems, Vol 11, No. 4, 1989.
[6] W J Stoddart and F Zeyda. A Unification of Probabilistic Choice within a Design-based Model of Reversible Computation. Formal Aspect of Computing, Published on-line 2007. Accepted for publication in the Special Issue on Unifying Theories of Programming, DOI 10.1007/s00165-007-0048-1.
[7] W J Stoddart, F Zeyda, and A R Lynas. A Design-based model of reversible computation. In UTP'06, First International Symposium on Unifying Theories of Programming, volume 4010 of Lecture Notes in Computer Science, June 2006.
[8] W J Stoddart, F Zeyda, and A R Lynas. A reversible virtual machine. In Proceedings of Reversible Computation 2009, 2009.
[9] W J Stoddart, F Zeyda, and A R Lynas. A virtual machine for supporting reversible probabilistic guarded command languages. Electronic Notes in Theoretical Computer Science, 253, 2010. DOI information: 10.1016/j.entcs.2010.02.005.
[10] M. V. Zelkowitz. Reversible execution. Commun. ACM, 16(9), 1973.
[11] F Zeyda. Reversible Computations in B. PhD thesis, University of Teesside, 2007.